Stochastic Game Analysis and Latency Awareness for Self-Adaptation

JAVIER CÁMARA, Carnegie Mellon University
GABRIEL A. MORENO, Carnegie Mellon University
DAVID GARLAN, Carnegie Mellon University
BRADLEY SCHMERL, Carnegie Mellon University

Although different approaches to decision-making in self-adaptive systems have shown their effectiveness in the past by factoring in predictions about the system and its environment (e.g., resource availability), no proposal considers the latency associated with the execution of tactics upon the target system. However, different adaptation tactics can take different amounts of time until their effects can be observed. In reactive adaptation, ignoring adaptation tactic latency can lead to suboptimal adaptation decisions (e.g., activating a server that takes more time to boot than the transient spike in traffic that triggered its activation). In proactive adaptation, taking adaptation latency into account is necessary to get the system into the desired state to deal with an upcoming situation. In this paper, we introduce a formal analysis technique based on model checking of stochastic multiplayer games (SMGs) that enables us to quantify the potential benefits of employing different types of algorithms for self-adaptation. In particular, we apply this technique to show the potential benefit of: (i) considering adaptation tactic latency in proactive adaptation algorithms, and (ii) making available additional tactics in the repertoire employed to adapt a system. Our results show that factoring in tactic latency in decision making, not only improves the outcome of adaptation, but also enables algorithms to fully exploit the set of available tactics for adaptation. We also present an algorithm to do proactive adaptation that considers tactic latency, and show that it achieves higher utility than an algorithm that under the assumption of no latency is optimal.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Proactive adaptation, Stochastic multiplayer games, Latency

1. INTRODUCTION

When planning how to adapt, self-adaptive approaches typically focus on the qualities of the resulting system, such as performance, operating cost, and reliability [Gar-

This work is supported in part by awards W911NF-13-1-0154 from the Army Research Office and N000141310401 from the Office of Naval Research. Co-financed by the Foundation for Science and Technology via project CMU-PT/ ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN with COMPETE ref.: FCOMP-01-0124-FEDER-012983. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. (DM-0001685).
Author's addresses: J. Cámara, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, and G.A. Moreno, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, and David Garlan, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, and Bradley Schmerl, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213,

DOI:http://dx.doi.org/10.1145/0000000.0000000

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 OCT 2014	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Stochastic Game Analysis and Latency Awareness for Self-Adaptation	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
Gabriel Moreno Javier Camara; David Garlan; Bradley Schmerl	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**25**	

lan et al. 2004; Zhang and Lung 2010], or safety and liveness properties of the system [Braberman et al. 2013; Goldman et al. 2003]. However, properties of the adaptation itself are largely ignored. One such property is the time it takes for an adaptation to produce its intended effect. Different adaptation tactics take different amounts of time until their effects can be observed. For example, consider two tactics to deal with an increase in the load of a system: reducing the fidelity of the results (e.g., less resolution, fewer elements, etc.), and adding a computer to share the load. Adapting the system to produce results with less fidelity may be achieved quickly if it can be done by changing a simple setting in a component, whereas powering up an additional computer to share the load may take some time. We refer to the time it takes since a tactic is started until its effect is observed as *tactic latency*. Current approaches to decide how to self-adapt do not take the latency of adaptation tactics into account when deciding what tactic to enact. For reactive adaptation, the consequence of this limitation is that the system may decide to adapt in a way that takes longer than other alternatives to achieve a marginally better result. For proactive adaptation, considering tactic latency is necessary so that the adaptation can be started with the sufficient lead time to be ready in time.

In this paper, we explore the use of tactic latency information in the case of proactive self-adaptation. Specifically, the contribution of this paper is twofold:

(1) A novel analysis technique based on model checking of stochastic multiplayer games (SMGs) that enables us to quantify the potential benefits of employing different types of algorithms for self-adaptation. Specifically, we show how the technique can be used to: (i) compare alternatives that consider tactic latency information for proactive adaptation with those that are not latency-aware, and (ii) quantify the potential improvement in the outcome of adaptation of incorporating additional tactics in the adaptation model for the system.
(2) A specific latency-aware algorithm for proactive adaptation. The algorithm extends one computes the optimal sequence of adaptation decisions for anticipatory dynamic configuration [Poladian et al. 2007]. The algorithm presented in this paper considers the effects of latency on adaptation.

Our formal verification results show that factoring in tactic latency in decision making improves the outcome of adaptation both in worst and best-case scenarios. Moreover, results indicate that while non-latency-aware algorithms can prevent the selection of available tactics that could help improve the outcome of adaptation, latency-aware algorithms are able to better exploit adaptation tactic repertoires. This is consistent with the results obtained for our latency-aware proactive adaptation algorithm, showing that it is able to obtain higher utility than Poladian et al.'s algorithm, which is optimal under the assumption of no tactic latency.

The remainder of this paper is structured as follows: Section 2 summarizes Znn.com, the example used to illustrate our approach. Section 3 introduces some background and related work. Section 4 describes our technique for analyzing adaptation based on model checking of stochastic games. Next, section 5 presents our algorithm for latency-aware proactive adaptation. Finally, section 6 concludes the paper and indicates future research directions.

2. EXAMPLE

Znn.com [Cheng et al. 2009a] is a case study portraying a representative scenario for the application of self-adaptation in software systems which has been extensively used to assess different research advances in self-adaptive systems. Znn.com embodies the typical infrastructure for a news website, and has a three-tier architecture consisting

of a set of servers that provide contents from backend databases to clients via front-end presentation logic (Figure 1). The system uses a load balancer to balance requests across a pool of replicated servers, the size of which can be adjusted according to service demand. A set of clients makes stateless requests, and the servers deliver the requested contents.

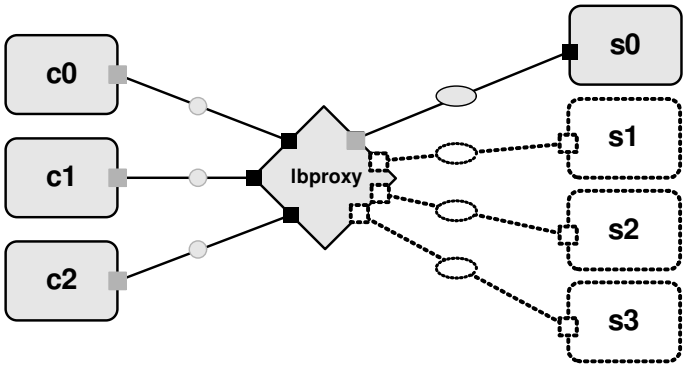

Fig. 1: Znn.com system architecture

The main objective for Znn.com is to provide content to customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. It is considered that from time to time, due to highly popular events, Znn.com experiences spikes in requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, the system can maintain functionality at a reduced level of fidelity by setting servers to return only textual content during such peak times, instead of not providing service to some of its customers. Concretely, there are three quality objectives for the self-adaptation of the system: *(i)* performance, which depends on request response time, server load, and network bandwidth, *(ii)* cost, which is associated with the number of active servers, and *(iii)* fidelity, which maps to the fidelity level of the contents being served (e.g., text, images).

In Znn.com, when response time becomes too high, the system is able to increment its server pool size if it is within budget to improve performance; or switch servers to textual mode if the cost is near to budget limit.

3. BACKGROUND AND RELATED WORK

This section first introduces the adaptation model that we assume in this paper. Next, we overview probabilistic model checking of SMGs, the technique upon which we build to analyze different kinds of adaptation in our approach. Finally, we present related work in proactive self-adaptation.

3.1. Adaptation Model

Although there are many approaches that rely on a closed-loop control approach to self-adaptation, including those that exploit architectural models for reasoning about the target system under management [Garlan et al. 2004; Kramer and Magee 2007; Oreizy et al. 1999], in this paper we use some of the high-level concepts in Rainbow [Garlan et al. 2004] as a reference framework to illustrate our approach. Rainbow is an architecture-based platform for self-adaptation, which has among its distinct features an explicit architecture model of the target system, a collection of adaptation tactics, and utility preferences to guide adaptation.

We assume a model of adaptation that represents adaptation knowledge using the following high-level concepts:[1]

— **Tactic:** is a primitive action that corresponds to a single step of adaptation, and has an associated: (i) cost/benefit impact on the different quality dimensions, and (ii) latency, which corresponds to the time it takes since a tactic is started until its effect is observed.[2] For instance, in Znn.com we can specify pairs of tactics with opposing effects for enlisting/discharging servers, or increasing/reducing the fidelity of the contents being served.

— **Utility Profile:** To enable the selection of tactics at run-time, we assume that adaptation is driven by utility functions and preferences, which are sensitive to the context of use and able to consider trade-offs among multiple potentially conflicting objectives. The different qualities of concern are characterized as utility functions that map them to architectural properties. In this case, we assume that utility functions are defined by an explicit set of value pairs (with intermediate points linearly interpolated).
Table I summarizes the utility functions for Znn.com. Function U_R maps low response times (up to 100ms) with maximum utility, whereas values above 2000ms are highly penalized (utility below 0.25), and response times above 4000ms provide no utility. Function U_F maps a low (1) level of content fidelity (e.g., textual version of contents) to a utility 0.5, whereas a high level (2) of content fidelity (e.g., including images/video) is mapped to maximum utility. Function U_C maps a increasing cost (derived from the number of active servers) to lower utility values. Utility preferences capture business preferences over the quality dimensions, assigning a specific weight (w_{U_R}, w_{U_F}, w_{U_C}) to each one of them. In the context of Znn.com, preference is typically given to performance over cost and fidelity.

By evaluating how different tactic execution sequences might affect the different qualities of concern using a utility profile, a proactive adaptation algorithm can build a strategy with the objective of maximizing accrued utility throughout the execution of the system.

Table I: Utility functions and preferences for Znn.com

U_R			U_F	U_C	
0 : 1.00	500 : 0.90	2000 : 0.25	1 : 0.50	0 : 1.00	3 : 0.30
100 : 1.00	1000 : 0.75	4000 : 0.00	2 : 1.00	1 : 1.00	4 : 0.00
200 : 0.99	1500 : 0.50			2 : 0.90	

3.2. Model Checking Stochastic Games

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains that range from power management or wireless communication protocols, to biological systems. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic behavior, effectively enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system's use of resources, time, etc.).

Competitive behavior may also appear in (stochastic) systems when some component cannot be controlled, and could behave according to different or even conflicting

[1] We use a simplified version of Stitch [Cheng and Garlan 2012] to illustrate the main ideas in this paper.
[2] Stitch incorporates a different notion of timing delay to monitor the outcome of tactic executions in reactive adaptation strategies, which is not discussed in this paper.

goals with respect to other components in the system. In such situations, a natural fit is modeling a system as a game between different players, adopting a game-theoretic perspective. Automatic verification techniques have been successfully used in this context, for instance for the analysis of security [Kremer and Raskin 2001] or communication protocols [Hoek and Wooldridge 2003].

Our approach to analyzing adaptation builds upon a recent technique for modeling and analyzing stochastic multi-player games (SMGs) extended with rewards [Chen et al. 2013a]. In this approach, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic.

Definition 3.1 (*SMG*). A turn-based stochastic multi-player game augmented with rewards (SMG) is a tuple $\mathcal{G} = \langle \Pi, S, A, (S_i)_{i\in\Pi}, \Delta, AP, \chi, r\rangle$, where Π is a finite set of players; $S \neq \emptyset$ is a finite set of states; $A \neq \emptyset$ is a finite set of actions; $(S_i)_{i\in\Pi}$ is a partition of S; $\Delta : S \times A \to \mathcal{D}(S)$ is a (partial) transition function; AP is a finite set of atomic propositions; $\chi : S \to 2^{AP}$ is a labeling function; and $r : S \to \mathbb{Q}_{\geq 0}$ is a reward structure mapping each state to a non-negative rational reward. $\mathcal{D}(X)$ denotes the set of discrete probability distributions over finite set X.

In each state $s \in S$ of the SMG, the set of available actions is denoted by $A(s) = \{a \in A \mid \Delta(s, a) \neq \bot\}$. We assume that $A(s) \neq \emptyset$ for all states s in the model. Moreover, the choice of which action to take in every state s is under the control of a single player $i \in \Pi$, for which $s \in S_i$. Once action $a \in A(s)$ is selected by a player, the successor state is chosen according to probability distribution $\Delta(s, a)$.

Definition 3.2 (*Path*). A path of SMG $\mathcal{G}$ is an (in)finite sequence $\lambda = s_0 a_0 s_1 a_1 \ldots$ ***s.t.*** $\forall j \in \mathbb{N} \bullet a_j \in A(s_j) \wedge \Delta(s_j, a_j)(s_{j+1}) > 0$. The sets of all finite paths in $\mathcal{G}$ is denoted as $\Omega^+_{\mathcal{G}}$.

Players in the game can follow strategies for choosing actions in the game, cooperating with each other in coalition to achieve a common goal, or competing to achieve their own (potentially conflicting) goals.

Definition 3.3 (*Strategy*). A strategy for player $i \in \Pi$ in $\mathcal{G}$ is a function $\sigma_i : (SA)^* S_i \to \mathcal{D}(A)$ which, for each path $\lambda \cdot s \in \Omega^+_{\mathcal{G}}$ where $s \in S_i$, selects a probability distribution $\sigma_i(\lambda \cdot s)$ over $A(s)$.

In the context of this paper, we always refer to player strategies σ_i that are *memoryless* (i.e., $\sigma_i(\lambda \cdot s) = \sigma_i(\lambda' \cdot s)$ for all paths $\lambda \cdot s$, $\lambda' \cdot s \in \Omega^+_{\mathcal{G}}$), and *deterministic* (i.e., $\sigma_i(\lambda \cdot s)$ is a Dirac distribution for all $\lambda \cdot s \in \Omega^+_{\mathcal{G}}$). Memoryless, deterministic strategies resolve the choices in each state $s \in S_i$ for player $i \in \Pi$, selecting actions based solely on information about the current state in the game. These strategies are guaranteed to achieve optimal expected rewards for the kind of cumulative reward structures that we use in our models.[3]

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a property in a logic called rPATL. Concretely, rPATL can be used for expressing quantitative properties of stochastic multi-player games, and extends the logic PATL [Chen and Lu 2007] (a probabilistic version of ATL [Alur et al. 2002], a logic extensively used in multi-player games and multi-agent systems to reason about the ability of a set of players to collectively achieve a particular goal). Properties written

[3]See Appendix A.2 in [Chen et al. 2013a] for details.

in rPATL can state that a coalition of players has a strategy which can ensure that the probability of an event's occurrence or an expected reward measure meet some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C\rangle\rangle$ of ATL [Alur et al. 2002], combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [Bianco and de Alfaro 1995]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie \mathsf{x}}$ from [Forejt et al. 2011] to reason about goals related to rewards. An example of typical usage combining coalition and reward operators is $\langle\langle\{1,2\}\rangle\rangle\mathsf{R}^{\mathsf{r}}_{\geq 5}[\mathsf{F}^{\star}\phi]$ [4], meaning that "players 1 and 2 have a strategy to ensure that the reward r accumulated along paths leading to states satisfying state formula ϕ is at least 5, regardless of the strategies of other players." Moreover, extended versions of the rPATL reward operator $\langle\langle C\rangle\rangle\mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\star}\ \phi]$ and $\langle\langle C\rangle\rangle\mathsf{R}^{\mathsf{r}}_{\mathsf{min}=?}[\mathsf{F}^{\star}\ \phi]$, enable the quantification of the maximum and minimum accumulated reward r along paths that lead to states satisfying ϕ that can be guaranteed by players in coalition C, independently of the strategies followed by the rest of players.

In this paper, we employ rPATL specifications based on the extended versions of the reward operator to compute accrued utility rewards during system execution, enabling the analysis of worst- and best-case scenarios for different types of proactive adaptation algorithms.

3.3. Related Work

Poladian et al. demonstrated that when there is an adaptation cost or penalty, proactive adaptation outperforms reactive adaptation [Poladian et al. 2007]. Intuitively, if there is no cost associated with adaptation, a reactive approach could adapt at the time a condition requiring adaptation is detected without any negative consequence. In their work, Poladian et al. presented two algorithms for proactive adaptation that considered the penalty of adaptation when deciding how to adapt. One of the algorithms assumed perfect predictions of the environment, while the other handled uncertainty. The latter was used to improve self-adaptation in Rainbow [Cheng et al. 2009b], where Cheng et al. considered tactic latency only to skip the adaptation if the condition that triggered it was predicted to go away by itself before the adaptation tactic completed. However, the approach did not consider all the effects that arise due to tactic latency (see Section 5).

Proactive adaptation has received considerable attention in the area of service-based systems [Calinescu et al. 2011; Hielscher et al. 2008; Metzger et al. 2013; Wang and Pazat 2012] because of their reliance on third-party services whose quality of service (QoS) can change over time. In that setting, when a service failure or a QoS degradation is detected, a penalty has already been incurred, for example, due to service-level agreement (SLA) violations. Thus, proactive adaptation is needed to avoid such problems. Hielscher et al. proposed a framework for proactive self-adaptation that uses online testing to detect problems before they happen in real transactions, and to trigger adaptation when tests fail [Hielscher et al. 2008]. Wang and Pazat use online prediction of QoS degradations to trigger preventive adaptations before SLAs are violated [Wang and Pazat 2012]. These approaches ignore the adaptation latency.

Musliner considers adaptation time by imposing a limit on the time to synthesize a controller for real-time autonomous systems [Musliner 2001]. However, in that work there are not distinct planning and execution phases, and thus there is no considera-

[4]The variants of $\mathsf{F}^{\star}\phi$ used for reward measurement in which the parameter $\star \in \{0, \infty, c\}$ indicate that, when ϕ is not reached, the reward is zero, infinite or equal to the cumulated reward along the whole path, respectively.

tion of the latency of the different actions the system could take to adapt. In the area of dynamic capacity management for data centers, the work of Gandhi et al. considers the setup time of servers, and is able to deal with unpredictable changes in load by being conservative about removing servers when the load goes down [Gandhi et al. 2012]. Their work is specifically tailored to adding and removing servers to a dynamic pool, a setting that resembles the running example we use in this paper. However, their work assumes the environment is unpredictable, and, consequently, does not consider the possibility of being able to predict a reduction or transient spikes in load. Our approach, on the other hand, can exploit predictions of such events and either adapt as soon as possible when removing servers, or avoid adaptations completely.

4. ANALYZING ADAPTATION

This section describes our approach to analyze self-adaptation, based on model checking of SMGs. In a nutshell, the underlying idea behind the approach is modeling both the self-adaptive system and its environment as two players of a SMG, in which the system attempts to maximize an accrued reward (in the context of this paper, accrued utility during system execution). Although in general, the environment does not have any predefined goal, it is useful to consider it either as an adversary of the system, or as a cooperative player to enable worst- and best-case scenario analysis, respectively, of different classes of adaptation algorithms (e.g., latency-aware *vs.* non-latency-aware).

By expressing properties that enable us to quantify the maximum and minimum rewards that a player can achieve, independently of the strategy followed by the rest of players, we can analyze the performance of a particular type of adaptation algorithm, giving an approximation of the reward that an optimal decision maker would be able to guarantee both in worst- and best-case scenarios (by synthesizing strategies that optimize different rewards). These properties follow the general pattern $\langle\langle \mathsf{P} \rangle\rangle \mathsf{R}^{\mathsf{U}}_{\bowtie}[\mathsf{F}^{\mathsf{c}}\omega]$, where P is a set of players that can include the system and/or the environment, U is a reward that encodes the instantaneous utility of the system, $\bowtie \in \{\mathsf{min}=?, \mathsf{max}=?\}$ identifies whether we are considering the minimum or the maximum utility reward, respectively, and ω is a state formula that encodes a stop condition for the system's execution. Section 4.2 details how such properties are used in our approach.

In the remainder of this section, we first present a SMG model of Znn.com that enables the comparison of latency-aware against non-latency-aware adaptation. We then describe how these models can be analyzed and show some results for different instances of the model.

4.1. SMG Model

Our formal model is implemented in PRISM-games [Chen et al. 2013b], an extension of the probabilistic model-checker PRISM [Kwiatkowska et al. 2011] for modeling and analyzing SMGs. Our game is played in turns by two players that are in control of the behavior of the environment and the system, respectively. The SMG model consists of the following parts:

4.1.1. Player definition. Listing 1 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the environment module), whereas player sys controls all transitions that belong to the target_system module.[5] Global variable turn in line 4 is used to make players alternate, ensuring that for every state of the model, only one player can take action. Turn-based gameplay suffices to naturally model the interplay be-

[5]Actions enlist_trigger, enlist, and discharge are explicitly labeled to improve readability (see Listing 3), but are still asynchronous in our model.

tween the environment and the system, which only senses environment information and reacts to it if necessary at discrete time points.

```
1  player env environment endplayer
2  player sys target_system,[enlist],[enlist_trigger],[discharge], [decrease_f], [increase_f] endplayer
3  const ENV_TURN=1, SYS_TURN=2;
4  global turn:[ENV_TURN..SYS_TURN] init ENV_TURN;
```

Listing 1: Player definition for Znn.com's SMG

4.1.2. Environment. The environment is in control of the evolution of time and other variables of the execution context that are out of the system's control (e.g., service requests arriving at the system). The choices in the environment module are specified non-deterministically to obtain a representative specification of the environment (through strategy synthesis) that is not limited to specific behaviors, since this would limit the generality of our analysis. Listing 2 shows the encoding used for the environment, in which Lines 1-3 define different constants that parameterize its behavior:[6]

— MAX_TIME defines the time frame for the system's execution in the model ([0,MAX_TIME]).
— TAU sets time granularity, defining the frequency with which the environment updates the value of non-controllable variables, and the system responds to these changes. The total number of turns for both players in the SMG is MAX_TIME/TAU. Note that two consecutive turns of the same player are separated by a time period of duration TAU.
— MAX_ARRIVALS constrains the maximum total number of requests that can arrive at the system for processing throughout its execution. Unconstrained arrivals would result in an unrealistic behavior of the environment (e.g., by following the strategy of continuously flooding the system with requests).
— MAX_INST_ARRIVALS is the maximum number of arrivals that the environment can place for the system to process during its turn (i.e., during one TAU time period).

```
1  const MAX_TIME;
2  const TAU;
3  const MAX_ARRIVALS, MAX_INST_ARRIVALS;
4
5  module environment
6  t : [0..MAX_TIME] init 0;
7  arrivals_total : [0..MAX_ARRIVALS] init 0;
8  arrivals_current : [0..MAX_INST_ARRIVALS] init 0;
9  a_upd : bool init false;
10   [] (t<MAX_TIME) & (turn=ENV_TURN) & (arrivals_total+x<MAX_ARRIVALS) & (!a_upd) -> (arrivals_current'=x) &
         (a_upd'=true);
11   ...
12   [] (t<MAX_TIME) & (turn=ENV_TURN) & (a_upd) -> 1:(t'=t+TAU) & (a_upd'=false) &
         (arrivals_total'=arrivals_total+arrivals_current) & (turn'=SYS_TURN);
13 endmodule
```

Listing 2: Environment module

[6]Constant values not defined in the model are provided as command-line input parameters to the tool.

Moreover, lines 6-9 declare the different variables that define the state of the environment:

— t keeps track of execution time.
— arrivals_total keeps track of the accumulated number of arrivals throughout the execution.
— arrivals_current is the number of request arrivals during the current time period.

Each turn of the environment consists of two steps:

(1) Setting the amount of request arrivals for the current time period. This is achieved through a set of commands that follow the pattern shown in Listing 2, line 10: the guard in the command checks that (i) it is the turn of the environment, (ii) the end of the time frame for execution has not been reached yet, and (iii) the value of request arrivals for the current time period has not been set yet (controlled by flag a_upd). If the guard is satisfied, the command sets the value of request arrivals for the current time period (represented by x in the command). It is worth noticing that there may be as many of these commands as different possible values can be assigned to the number of request arrivals for the current time period (including zero for no arrivals). Probabilities in these commands are left unspecified, since it will be up to the strategy followed by the player (to be synthesized based on an rPATL specification) to provide the discrete probability distribution for this set of commands.
(2) Updating the values of the different environment variables (line 12), by: (i) increasing the t time variable one step, and (ii) adding the number of request arrivals for the current time period to the accumulator arrivals_total. In addition, the turn of the environment player finishes when this command is executed, since it modifies the value of variable turn, yielding the control of the game to the system player.

4.1.3. System. Module target_system (Listing 3) models the behavior of the target system (including the execution of tactics upon it), and is parameterized by the constants:

— MIN_SERVERS and MAX_SERVERS, which specify the minimum and maximum number of active servers that a valid system configuration can have.
— INIT_SERVERS is the number of active servers that the system has in its initial configuration.
— MIN_FIDELITY and MAX_FIDELITY, which specify the minimum and maximum fidelity levels for served content.
— INIT_FIDELITY is the level of fidelity of served content in the system's initial configuration.
— ENLIST_LATENCY is the latency of the tactic for enlisting a server, measured in number of time periods (i.e., the real latency for the tactic in time units is TAU * ENLIST_LATENCY). In our model, tactic latencies are always limited to multiples of the time period duration.
— MAX_RT and INIT_RT, which specify the system's maximum and initial response times, respectively.

Moreover, the module includes variables which are relevant to represent the current state of the system:

— s corresponds to the number of active servers.
— f is the fidelity level of the contents being served (in this version of Znn.com, we consider it to be the same for all servers).
— rt is the system's response time.

```
1  const MIN_SERVERS, MAX_SERVERS, INIT_SERVERS;
2  const MIN_FIDELITY, MAX_FIDELITY, INIT_FIDELITY;
3  const ENLIST_LATENCY;
4  const MAX_RT, INIT_RT;
5
6  module target_system
7  s : [0..MAX_SERVERS] init INIT_SERVERS;
8  f : [1..MAX_FIDELITY] init INIT_FIDELITY;
9
10 rt : [0..MAX_RT] init INIT_RT;
11 counter:[−1..ENLIST_LATENCY] init −1;
12   [] (s<=MAX_SERVERS) & (turn=SYS_TURN) & (counter!=0) −> (counter'=counter>0?counter−1:counter) &
         (turn'=ENV_TURN) & (rt'=totalTime);
13   [enlist_trigger] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=−1) −> (counter'=ENLIST_LATENCY) &
         (turn'=ENV_TURN) & (rt'=totalTime);
14   [enlist] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=0) −> 1: (s'=s+1) & (counter'=−1) & (turn'=ENV_TURN) &
         (rt'=totalTime);
15   [discharge] (s>MIN_SERVERS) & (turn=SYS_TURN) & (counter!=0) −> (s'=s−1) &
         (counter'=counter>0?counter−1:counter) & (turn'=ENV_TURN) & (rt'=totalTime) ;
16   [decrease_f] (turn=SYS_TURN) & (f>MIN_FIDELITY) & (counter!=0)−> (turn'=ENV_TURN) & (f'=f−1) &
         (counter'=counter>0?counter−1:counter);
17   [increase_f] (turn=SYS_TURN) & (f<MAX_FIDELITY) & (counter!=0)−> (turn'=ENV_TURN) & (f'=f+1) &
         (counter'=counter>0?counter−1:counter);
18 endmodule
```

Listing 3: System module

— counter is used to control the delay between the triggering of a tactic and the moment in which it becomes effective in the target system. In this case, the variable is used to control the delay between the activation of a server, and the time instant in which it really becomes active.

During its turn, the system can decide not to execute any tactics, returning the turn to the environment player by executing the command defined in line 12, Listing 3. Alternatively, the system can execute one of these tactics:

— Activation of a server, which is carried out in two steps:
  (1) Triggering of activation through the execution of the command labeled as enlist_trigger (line 13). This command only executes if the current number of active servers is less than the maximum allowed, and the counter that controls tactic latency is inactive (meaning that there is not currently a server already booting in the system). Upon execution, the command activates the counter by setting it to the value of the latency for the tactic, and returns the turn to the environment player.
  (2) Effective activation through the enlist command (line 14), which executes when the counter that controls tactic latency reaches zero, incrementing the number of servers in the system, and deactivating the counter. All the commands in this module, except for the latter, decrement the value of the counter 1 unit, if the counter is activated (counter'=counter>0?counter-1:counter).
— Deactivation of a server, which is achieved through the discharge command (line 15), which decrements the number of active servers. The command fires only if the current number of active servers is greater than the minimum allowed and the counter for server activation is not active.
— Lowering the fidelity of all active servers, setting them to textual mode through the execution of the command decrease_f (line 16). This tactic decreases the value of the fidelity variable f, and thus increases the service rate, which in turn causes a reduction in the system's response time.

— Raising the fidelity of all active servers, setting them to multimedia mode through the execution of command increase_f (line 17), which has the opposite effect of decrease_f.

In addition, all the commands in this module update the value of the response time according to the request arrivals during the current time period and the number of active servers (computed using of an M/M/c queuing model [Chiulli 1999], encoded by formula totalTime). Moreover, note that the latency of all tactics in the model, except for the one to enlist servers, is zero.

4.1.4. Utility profile. Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of instantaneous utility. Specifically, formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences.

```
1  formula uR = (rt>=0 & rt<=100? 1:0)
2   +(rt>100&rt<=200?1+(−0.01)∗((rt−100)/(100)):0)
3   ...
4   +(rt>2000&rt<=4000?0.25+(−0.25)∗((rt−2000)/(2000)):0)
5   +(rt>4000 ? 0:0);
6  ...
7  const W_UR,W_UF,W_UC;
8  rewards "rIU"
9   (turn=SYS_TURN) : TAU∗(W_UR∗uR + W_UF∗uF +W_UC∗uC);
10 endrewards
```

Listing 4: Utility functions and reward structure

Listing 4 illustrates in lines 1-5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function U_R in the first column of Table I. The formula in the example computes the utility for performance, based on the value of the variable for system response time rt. Moreover, lines 8-10 show how a reward structure can be defined to compute a single utility value for any state by using utility preferences (in this case defined as weights W_UR, W_UC, and W_UF). Specifically, each state in which it is the turn of the system player is assigned with a reward corresponding to the entire elapsed time period of duration TAU, during which we assume that instantaneous utility does not change.

```
1  rewards "rEIU"
2   (turn=SYS_TURN) : TAU∗(W_UR∗uER + W_UF∗uF +W_UC∗uC);
3  endrewards
```

Listing 5: Expected utility reward structure

In latency-aware adaptation, the instantaneous real utility extracted from the system coincides with the utility expected by the algorithm's computations during the tactic latency period. However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance (i.e., the server is booting up, but not processing requests yet). To enable analysis of real *vs.* expected utility in non-latency-aware adaptation,

we add to the model a new reward structure that encodes expected instantaneous utility rEIU (Listing 5). In this case, the utility for performance during the latency period (encoded in formula uER) is computed analogously to uR in Listing 4, but based on the response time that the system would have with s+1 servers during the latency period.

4.2. Analysis

In order to compare latency-aware *vs.* non-latency-aware adaptation, we make use of rPATL specifications that enable us to analyze (i) the maximum utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario), and (ii) the maximum utility that adaptation is able to obtain under ideal environmental conditions (best-case scenario).

4.2.1. Latency-aware Adaptation. In latency-aware adaptation, the real adaptation extracted from the system coincides with the utility that the adaptation algorithm uses for decision making.

— **Worst-case scenario analysis.** We define the *real guaranteed accrued utility* (U_{rga}) as the maximum real instantaneous utility reward accumulated throughout execution that the system player is able to guarantee, independently of the behavior of the environment player:

$$U_{rga} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

This enables us to obtain the utility that an optimal self-adaptation algorithm would be able to extract from the system, given the most adverse possible conditions of the environment. Alternatively, U_{rga} can also be obtained by computing a strategy for the environment, based on the minimization of the same reward:

$$\langle\langle \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{min}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

— **Best-case scenario analysis.** To obtain the *real maximum accrued utility* achievable (U_{rma}), we specify a coalition of the system and environment players, which behave cooperatively to maximize the utility reward:

$$U_{rma} \triangleq \langle\langle \mathsf{sys}, \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

4.2.2. Non-latency-aware Adaptation. In the case of non-latency-aware adaptation, the real utility does not coincide with the expected utility that an arbitrary algorithm would employ for decision-making, therefore we need to proceed with the analysis in two stages:

(1) Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based on an rPATL specification that obtains the expected guaranteed accrued utility (U_{ega}) for the system player:

$$U_{ega} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rEIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

For the specification of this property we employ the expected utility reward rEIU (Listing 5) instead of the real utility reward rIU. Moreover, it is worth observing that for latency-aware adaptation $U_{ega} = U_{rga}$.

(2) Verify the specific property of interest (e.g., U_{rga}, U_{rma}) under the generated strategy. We do this by using PRISM-games to build a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, once we have computed a strategy for the system player to maximize expected utility, we quantify the reward for real utility in the new game in which the system player strategy has already been fixed.

4.3. Results

In this section, we first compare worst- and best-case scenario analysis of a version of Znn.com that includes only the pair of tactics to enlist/discharge servers that are affected by latency in order to compare latency-aware and non-latency aware adaptation. Second, we provide some results to quantify the impact in utility that adding tactics to increase/reduce content fidelity introduce in the system. The improvement of introducing the new tactics is shown both in the case of latency-aware and non-latency-aware adaptation.

4.3.1. Comparing Latency-aware vs. Non-Latency-aware Adaptation. Table II compares the results for the utility extracted from the system by a latency-aware *vs.* a non-latency-aware version of the system player, for two different models of Znn.com that represent an execution of the system during 100 and 200s, respectively. The models consider a pool of up to 4 servers, out of which 2 are initially active, and includes a repertoire of tactics limited to enlisting/discharging servers. The period duration TAU is set to 10s, and for each version of the model, we compute the results for three variants with different latencies for the activation of servers of up to 3*TAU s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s. The fidelity level in this set of experiments is fixed, therefore we factor it out of the utility calculation ($w_{\mathsf{U_R}} = 0.6$, $w_{\mathsf{U_F}} = 0$, $w_{\mathsf{U_C}} = 0.4$).

We define the delta between the expected and the real guaranteed utility as:

$$\Delta U_{er} = (1 - \frac{U_{ega}}{U_{rga}}) \times 100$$

Moreover, we define the delta in real guaranteed utility between latency-aware an non-latency aware adaptation as:

$$\Delta U_{rga} = (1 - \frac{U^n_{rga}}{U^l_{rga}}) \times 100,$$

where U^n_{rga} and U^l_{rga} designate the real guaranteed accrued utility for non-latency-aware and latency-aware adaptation, respectively. The delta in real maximum accrued utility (ΔU_{rma}) is computed analogously to ΔU_{rga}.

Table II: SMG model checking results for Znn.com

MAX_TIME (s)	Latency (s)	Latency-Aware				Non-Latency-Aware				ΔU_{rga} (%)	ΔU_{rma} (%)
		U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{rma}	U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{rma}		
100	TAU	53.77	53.77	0	99.6	65.97	48.12	-27.05	79.99	10.5	19.68
	2*TAU	49.35	49.35	0	99.6	64.3	42.1	-34.5	78.39	14.69	21.29
	3* TAU	45.6	45.6	0	99.6	64.3	33.25	-48.2	78.39	27	21.29
200	TAU	110.02	110.02	0	199.6	127.25	95.9	-24.63	156.79	12.83	21.44
	2*TAU	105.6	105.6	0	199.6	125.57	76.6	-38.99	155.19	27.46	22.24
	3* TAU	101.17	101.17	0	199.6	123.9	66.15	-46.6	153.59	34.61	23.05

Table II shows that latency-aware adaptation outperforms in all cases its non-latency-aware counterpart. In the worst-case scenario, latency-aware adaptation is able to guarantee an increment in utility extracted from the system, independently of the behavior of the environment (ΔU_{rga}) that ranges between approximately 10 and 34%, increasing progressively with higher tactic latencies. In the best-case scenario (cooperative environment), the maximum utility that latency-aware adaptation can achieve does not experience noticeable variation with latency, staying in the range 19-23% in all cases. Regarding the delta between expected and real utility that adaptation can guarantee, we can observe that ΔU_{er} is always zero in the case of latency-aware

adaptation, since expected and real utilities always have the same value, whereas in the case of non-latency-aware adaptation there is a remarkable decrement that ranges between 24 and 48%, also progressively increasing with higher tactic latency.

4.3.2. Quantifying the Impact of Tactics on Utility. In this section, we compare the results for the utility extracted from the system for the worst-case scenario, using four different model variants of Znn.com. Two of the variants correspond to the latency-aware adaptation case when it includes: (i) only the pair of tactics to enlist/discharge servers (LA), and (ii) an extended set of tactics that include the tactics to enlist/discharge servers, plus the pair of tactics to increase/reduce content fidelity (LA+). The other two variants include the same sets of tactics for the non-latency-aware adaptation case (indicated by NLA and NLA+, respectively).

All models represent an execution of the system during 1000s, and consider a pool of up to 4 servers, out of which 2 are initially active. The period duration TAU is set to 10s, and for each version of the model, we compute the results of a latency range for the activation of servers between 0 and 7*TAU s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s for high fidelity, and 0.9s for low fidelity. The utility preferences used for the experiments give preference to performance over cost and fidelity ($w_{\mathsf{U_R}} = 0.6$, $w_{\mathsf{U_F}} = 0.2$, $w_{\mathsf{U_C}} = 0.2$).

— **Latency-aware Adaptation**. Figure 2 compares the two variants of latency-aware adaptation. In the LA variant, it can be observed that the progressive increment in latency of the enlist server tactic results in a proportional reduction of the real guaranteed utility U_{rga}. However, for increasing latency values in the LA+ variant, U_{rga} only decreases moderately in comparison with the LA variant, due to the fact that the optimal strategy synthesis algorithm starts favoring the selection of the tactic to reduce fidelity instead of the one to enlist a new server. Although decreasing fidelity has a negative influence in the value of the fidelity utility $\mathsf{U_F}$ and has a moderate impact in reducing response time compared to adding a new server, increasing latencies of the enlist server tactic dwindle its capability to extract system utility over time, compared to reducing content fidelity.

— **Non-latency-aware Adaptation**. Figure 3 compares the two variants of non-latency-aware adaptation. In contrast with the latency-aware adaptation case, there is a clear discrepancy between real guaranteed utility U_{rga}, and the expected guaranteed utility U_{ega} both for the NLA and NLA+ variants. Interestingly, it can be observed how the addition of the pair of fidelity tactics does not represent any difference in U_{ega} nor U_{rga} between the NLA and NLA+ variants. This is a direct consequence of the strategy synthesis process not being aware of the latency of the tactic to enlist servers. Concretely, the synthesis only considers the positive net impact on utility of enlisting a server, which outweighs the impact on utility of reducing fidelity, effectively preventing the selection of the fidelity reduction tactic, independently of the fact that it is capable to extract more system utility over time than the tactic to enlist servers.

5. LATENCY-AWARE ADAPTATION

Latency-aware adaptation takes into account the tactics' latency when deciding how to adapt. In our approach, the goal is to consider the latency of the tactics so that the sum of utility provided by the system over time is maximized. The effect of tactic latency on utility is that for tactics that have some latency, the system does not start to accrue the utility gain associated with the tactic until some time after the enactment of the tactic. Moreover, negative impacts of the tactic may have no latency, and start without delay. For example, when adding a server to the system, the server takes some time to

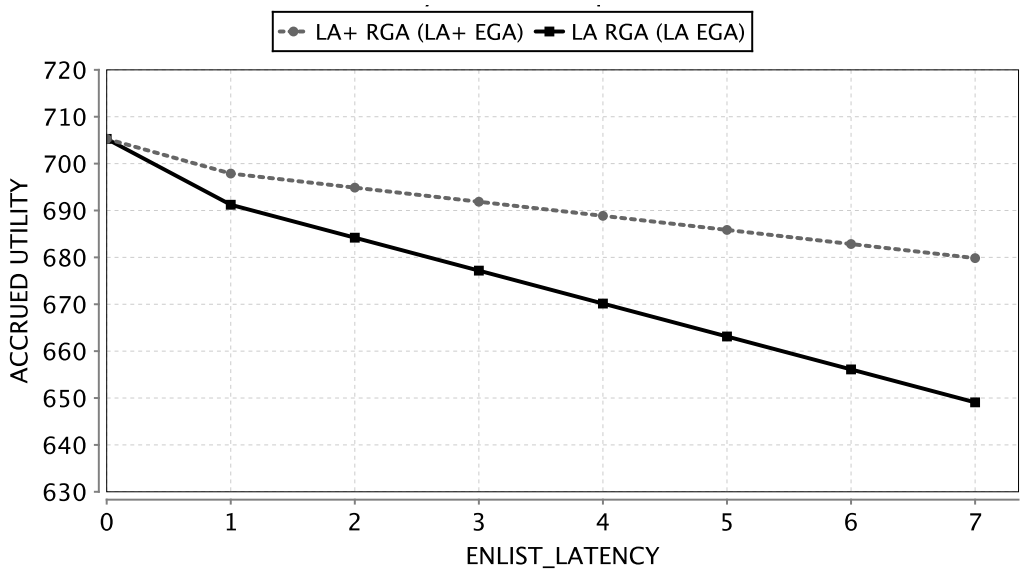

Fig. 2: Fidelity tactics impact on utility: Latency-Aware Adaptation

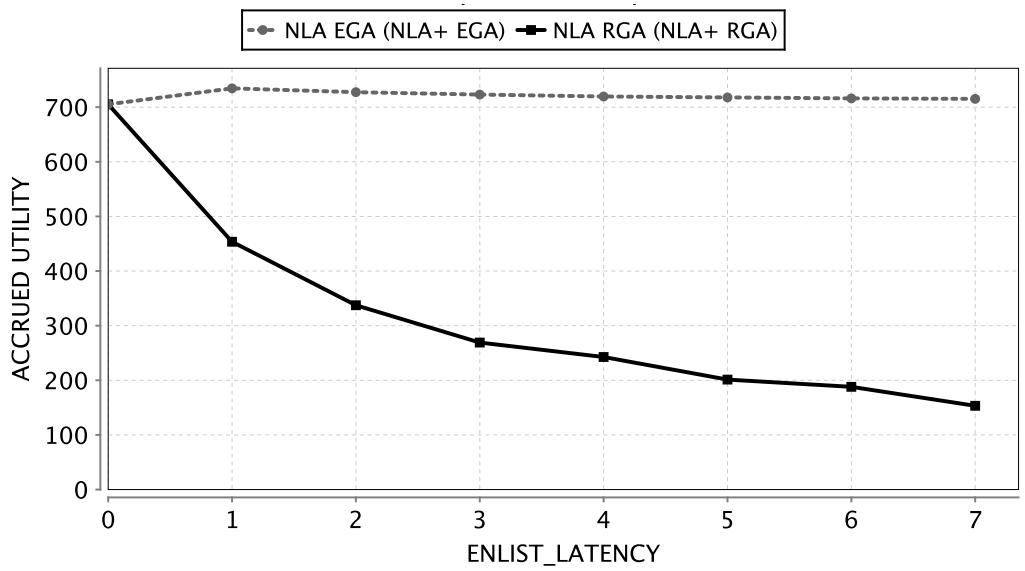

Fig. 3: Fidelity tactics impact on utility: Non-Latency-Aware Adaptation

boot and be online, whereas it starts consuming power—and thereby increases cost—immediately. In this example, it means that the tactic to add a server causes a drop in utility before it results in a gain.

Another consequence of tactic latency is that some near-future system configurations can be infeasible. For example, let us suppose that the system has to deal with an increase in load within 5 seconds, and it could handle that with an additional server. If enlisting an additional server takes 10 seconds, then the desired configuration that has

one additional server 5 seconds into the future is infeasible. Current approaches that do not take latency into account would consider that solution regardless of whether it is feasible or not. When proactively looking ahead, taking adaptation latency into account allows the adaptation mechanism to rule out infeasible configurations from the adaptation space.

A complication arises when tactic latency is longer than the interval between adaptation decisions. When that is the case, it is possible that during an adaptation decision, a tactic that has been previously started has not yet reached the point where its effect will have been realized. If the decisions are made based only on the currently observed state of the system, ignoring the expected effect of adaptations in progress, the system will overcompensate, starting unnecessary adaptations. What is needed is a model of the system that not only represents the current state of the system, but also keeps track of the expected state of the system in the near future based on the tactics that have been started but have not yet completed.

5.1. Algorithm

The algorithm we present is an extension of an algorithm developed by Poladian et al. to compute the optimal sequence of adaptation decisions for anticipatory dynamic configuration [Poladian et al. 2007]. Using dynamic programming and relying on a perfect prediction of the environment for the duration of a system run, their algorithm can find the adaptation decision that at each time step maximizes the future aggregate utility, while accounting for the penalty of switching configurations. They showed that the algorithm had pseudo-polynomial time complexity, and was therefore suitable for online adaptation.

The key improvement our algorithm brings is how the latency of tactics is taken into account. On the one hand, there is an adaptation cost that latency induces. For example, if adding a server takes λ seconds from the time a server is powered up until it can start processing requests, and ΔU_c is the additional cost the new server incurs, then the adaptation cost is $\lambda \Delta U_c$. This cost could be partially handled by the original algorithm, as a reconfiguration penalty. However, that is not sufficient to handle the other issues previously mentioned that latency brings, namely, the infeasibility of configurations and the need to track adaptation progress. Our algorithm for latency-aware proactive adaptation explicitly handles the issues that arise due to tactic latency.

The algorithm requires iterating over all the possible configurations of the system, where a configuration describes variable aspects of the system relevant to the adaptation decision. In the Znn example, a configuration indicates how many servers are in the pool of servers, and what is the fidelity level of the content being served. To keep track of adaptation progress, a configuration also encodes information about the progress of adaptations that have non-zero latency. In our example, that means that a configuration indicates whether a new server is being added, and how much progress that tactic has made. It is important to note that the information about progress is only needed at the granularity of the evaluation period τ. In general, C is the set of possible configurations, and C_i is the ith configuration, for $i \in \{1 \dots |C|\}$. For our running example, $C = (S \times A \times F) \setminus \{(s, a, f) : S \times A \times F | s = 4 \wedge a \in A \setminus \{0\}\}$, where $S = \{1 \dots 4\}$ is the number of active servers in the system; $A = \left\{0 \dots \lceil \frac{\lambda}{\tau} \rceil\right\}$ is the number of evaluation periods until the addition of a server completes, with 0 indicating that the tactic is not being executed; and $F = \{1, 2\}$ is the fidelity level. Since the tactic to add a server cannot be used when the system already has the maximum number of servers, all the configurations with 4 servers and the tactic running are not included in C.

The algorithm also needs to determine whether a particular configuration can be reached at a particular time, and tactic latency plays a key role in that de-

termination. More specifically, the algorithm needs to determine if configuration c' can be reached from configuration c in one evaluation period—the boolean function $isReachableFromConfig(c, c')$ encapsulates that. In addition, it needs to know if configuration c' can be reached at the current time—the function $isReachableNow(c')$ determines that. In addition to latency, blocking effects between tactics are also considered by these functions. For example, in our running example, only one tactic can be used in an evaluation period. More details about these two functions are provided in section 5.1.1.

In reactive adaptation, the decision algorithm is typically invoked upon events that require an adaptation to be performed. However, for proactive adaptation, the decision must be done periodically, looking ahead for future states that may require the system to adapt. Our algorithm is therefore run periodically, with a constant interval τ between runs. We limit the look-ahead of the algorithm to a near-term horizon of H evaluation periods, which in turn limits how far into the future the environment state needs to be estimated.[7] The estimation of the future environment state is accessed by the algorithm via the function $env(x)$, which returns the expected environment state x time units into the future.

Employing a dynamic programming approach, the algorithm (Algorithm 1) uses two matrices, u and n, to store partial solutions. The element $u_{i,t}$ holds the utility projected to be achieved from the evaluation period t (with $t = 0$ being the current period, $t = 1$ the next one, and so on) until the horizon if the system has configuration C_i at evaluation period t. An infeasible partial solution is marked by a value of $-\infty$ assigned to $u_{i,t}$. The element $n_{i,t}$ holds the configuration that the system must adopt in period $t+1$ to attain the projected utility $u_{i,t}$ if the configuration in period t is C_i.

The main loop (lines 1-23) works backwards from the horizon, computing the partial solutions using the partial solutions previously found. For each configuration (lines 2-22), it computes its projected utility or deems the configuration infeasible. For evaluation periods $t > 0$, all configurations are assumed feasible, and the only concern is whether one potential configuration is reachable from another potential configuration. However, for the current evaluation period ($t = 0$), only those configurations that can be reached are deemed feasible. The projected utility a configuration can achieve can achieve is the sum of the utility the configuration obtains in that particular evaluation period (line 6), and the maximum utility it can achieve in the periods after that. Computing the former relies on the function $U(c, e)$, which is the instantaneous utility provided by configuration c in environment e. To compute the latter, the algorithm iterates (lines 11-19) over all the feasible configurations that can follow (as determined by $isReachableFromConfig(C_i, C_j)$) to find the configuration that the system should have in evaluation period $t + 1$ to maximize the projected utility of having configuration C_i in evaluation period t (lines 14-17). Once all the possible solutions have been computed, the algorithm selects the configuration the system should have at the current time to maximize the projected utility (line 24). By comparing the current system configuration with the selected configuration along the different dimensions (S,A, and F in our example), it is easy to determine what adaptation tactics have to be started at the current time, if any.

5.1.1. Adaptation Feasibility. An important part of the proactive latency-aware adaptation algorithm is determining whether it is possible reach a particular system configuration through adaptation at a particular time in the near future. Obviously, tactic latency plays a fundamental role in this determination, not only because a tactic needed

[7]Environment state estimation is beyond the scope of our work, but techniques such as Poladian et al.'s calculus for combining multiple source of predictions [Poladian et al. 2007] can be used.

ALGORITHM 1: Latency-aware proactive adaptation

```
1: for t = H − 1 downto 0 do
2:   for i = 1 to |C| do
3:     u_{i,t} ← −∞ {assume infeasible configuration}
4:     n_{i,t} ← 0 {assume no next state}
5:     if t > 0 ∨ isReachableNow(C_i) then
6:       u_local ← τU(C_i, env(tτ))
7:       if t = H then
8:         u_{i,t} ← u_local
9:       else
10:        {find the next best configuration after i}
11:        for j = 1 to |C| do
12:          if u_{j,t+1} > −∞ ∧ isReachableFromConfig(C_i, C_j) then
13:            u_projected ← u_local + u_{j,t+1}
14:            if u_projected > u_{i,t} then
15:              u_{i,t} ← u_projected
16:              n_{i,t} ← j
17:            end if
18:          end if
19:        end for
20:      end if
21:    end if
22:  end for
23: end for
24: best ← arg max_i u_{i,0} {best starting configuration}
25: return C_best
```

to reach a configuration may take some time to execute, but also because it can block other tactics while executing. In our running example, only one tactic can be executed at a time, and, additionally, only one tactic can be started in each evaluation period. The algorithm uses two functions to determine the feasibility of possible adaptations.

The function $isReachableNow(c')$ returns *true* if it is possible to reach configuration c' immediately from the current configuration of the system. For example, if no tactic is executing in Znn, it would be possible to reach immediately a configuration in which the fidelity level has been changed, or one in which the tactic to add a new server has been started, but not both. On the other hand, if the tactic to add a server was executing (i.e., it was started in a previous evaluation period and has not completed yet), it would not be possible to reach any configuration other than the current one.

The function $isReachableFromConfig(c, c')$ returns *true* if configuration c' can be reached from configuration c in one evaluation period. More specifically, it assumes that *(i)* c will be the configuration at the beginning of the period, including the possible effect of tactics that could have been started at that time; *(ii)* one evaluation period will elapse allowing for example progress on a tactic with latency; and *(iii)* optionally a tactic can be started at the end of the period. For example, assuming that c is a configuration in which the tactic to add a server has one period left to complete, and c' is a configuration with one more active server and a different fidelity would be feasible because the tactic adding a server would complete in the elapsed period, and the fidelity can be changed immediately.

These two functions can be implemented in different ways as long as they satisfy their specification. Furthermore, since they are independent of the state of the environment, they can be computed offline, generating a lookup table to be used at runtime. Taking advantage of this, we used a Alloy [Jackson 2012] to formally specify system configurations, and adaptation tactics, and to compute the reachability functions offline. Alloy is a language based on first-order logic that allows modeling structures—known as signatures—and relationships between them in the form of constraints. One

advantage of using Alloy is that it is a declarative language, and, in contrast to imperative languages, only the effect of operations—tactics in our case—on the model must be specified, but not how the operations work. The Alloy analyzer can then be used to find structures that satisfy the model.

The basic definitions of the specification used to compute the reachability functions is shown in Listing 6. These definitions introduce the sets S, A, and C, representing the number of servers, the progress of the tactic to add a server, and the possible configurations the system can have. The elements of S and A are not numbers, but just abstract elements of an ordered set. It is possible to refer to the first and last element of the set that represents the possible levels of active servers as `servers/first` and `servers/last`, respectively. The relationships `prev` and `next` allow referring to the previous and next element in the ordered set. The signature C defines the set of all possible configurations. Without additional constraints, Alloy could generate elements of C with the same number of servers, progress of the tactic, and fidelity level. To force all elements of the set to be unique configurations, the constraint in lines 16-18 is introduced.

```
1  open util/ordering[S] as servers
2  open util/ordering[A] as progress
3  open util/boolean
4
5  sig S {} // the different number of active servers
6  sig A {} // the different levels of progress of the add server tactic
7
8  /* each element of C represents a configuration */
9  sig C {
10     s : S, // the number of active servers
11     a : A, // the progress of the add server tactic
12     f : Bool // fidelity level
13 }
14
15 // force all instances of C to be unique
16 fact uniqueInstances {
17     all disj c, c2 : C | !(c2.s = c.s and c2.a = c.a and c2.f = c.f)
18 }
```

Listing 6: Alloy model of configuration reachability: basic definitions

The specification of the tactics is shown in Listing 7. The tactic to add a server is decomposed into two predicates due to its latency. The first predicate (lines 1-6) specifies the start of the tactic. This predicate has two arguments c and c', representing the pre- and post-state, respectively. For the tactic to be able to start, it is required that no tactic is running, and that the configuration in the pre-state is not the last level of servers (i.e., the configuration has less than the maximum number of servers). In the post-state, the only change to the configuration is that the level of progress of the tactic is the first one. The tactic in the post-state has been started and will in subsequent steps go through all the levels of progress until it reaches the last one when it completes. The other predicate (lines 8-13) specifies how the configuration changes when the tactic makes progress in one evaluation period. The tactic can only make progress if it has not completed in the pre-state. In the post-state, the configuration will have the same fidelity level, and the next level of progress. If the latter is the last level of progress, then the tactic has completed and the post-state configuration has one more active server. Otherwise, the number of servers stays the same. The tactics for removing a server (lines 19-24) and for changing the fidelity (lines 26-31) do not have latency, and, therefore, do not need to be split into start and progress as the other

tactic. If no tactic is running, the system can just stay in the same configuration. In the model, the predicate at the end of Listing 7 is used to allow that behavior.

```
1  pred addServerTacticStart[c, c' : C] {
2      !tacticRunning[c] and c.s != servers/last
3      c'.a = progress/first
4      c'.s = c.s
5      c'.f = c.f
6  }
7
8  pred addServerTacticProgress[c, c' : C] {
9      c.a != progress/last
10     c'.a = progress/next[c.a]
11     c'.a = progress/last implies c'.s = servers/next[c.s] else c'.s = c.s
12     c'.f = c.f
13 }
14
15 pred tacticRunning[c : C] {
16     c.a != progress/last
17 }
18
19 pred removeServerTactic[c, c' : C] {
20     !tacticRunning[c] and c.s != servers/first
21     c'.s = servers/prev[c.s]
22     c'.a = c.a
23     c'.f = c.f
24 }
25
26 pred changeFidelityTactic[c, c' : C] {
27     !tacticRunning[c]
28     c'.s = c.s
29     c'.a = c.a
30     c'.f = Not[c.f]
31 }
32
33 pred noOp[c, c' : C] {
34     !tacticRunning[c]
35     c'.s = c.s
36     c'.a = c.a
37     c'.f = c.f
38 }
```

Listing 7: Alloy model of configuration reachability: tactics

Listing 8 defines the configuration reachability predicates. For `isReachableNow` (lines 2-4), configuration c' can be reached from c trivially if they are the same configuration, or if c' is the configuration resulting from starting a tactic when the system is in configuration c, and no passage of time is allowed. In the case of `isReachableFromConfig` (lines 7-9), configuration c' can be reached from c after one evaluation period if the configuration that results from letting one period to elapse, configuration $temp$, is such that configuration c' can be reached from $temp$ without any more passage of time. The predicate `timeStep` (lines 11-13) is used for the first part of this condition, and `isReachableNow` is reused for the second part.

The Alloy code in Listing 9 is used to generate the reachability functions. Each of the predicates will be used to generate the elements of the relationship `Result.reachable` for each of the reachability functions. The `run` commands in lines 13 and 15 are used to run the Alloy analyzer to generate the relationships that satisfy the corresponding predicates. The command specifies how many elements the solution should have in each set. For our example, when the latency of the tactic to add a server is 3τ, the solution must have 4 servers, $3\tau+1 = 4$ levels of progress for the tactic, and two fidelity

```
/* is c' reachable now if current config is c? */
pred isReachableNow[c, c' : C] {
    c = c' or removeServerTactic[c, c'] or addServerTacticStart[c, c'] or changeFidelityTactic[c, c']
}

/* is c' reachable from config c in one evaluation period? */
pred isReachableFromConfig[c, c' : C] {
    one temp : C | timeStep[c, temp] and isReachableNow[temp, c']
}

pred timeStep[c, c' : C] {
    noOp[c, c'] or addServerTacticProgress[c, c']
}
```

Listing 8: Alloy model of configuration reachability: functions

levels (not indicated in the command), for a total of 32 configurations.[8] Additionally, the run should produce exactly one result. The output of the Alloy analyzer can be exported to XML, and then transformed to a format suitable for its use at runtime.

```
sig Result {
    reachable : C->C
}

pred isReachableFromConfigGeneration {
    one r : Result | all c1,c2 : C | c1->c2 in r.reachable <=> isReachableFromConfig[c1,c2]
}

pred isReachableNowGeneration {
    one r : Result | all c1,c2 : C | c1->c2 in r.reachable <=> isReachableNow[c1,c2]
}

run isReachableFromConfigGeneration for exactly 4 S, exactly 4 A, exactly 32 C, exactly 1 Result

run isReachableNowGeneration for exactly 4 S, exactly 4 A, exactly 32 C, exactly 1 Result
```

Listing 9: Generation of configuration reachability tables in Alloy

5.2. Simulation

We implemented a simulation of a self-adaptive Znn with two goals. One was to evaluate the improvement that our algorithm for latency-aware (LA) proactive adaptation achieves compared to a non-latency-aware (NLA) approach. The second one, was to compare the theoretical results obtained with the SMG for generic NLA and LA algorithms with the results obtained with a concrete algorithm. Using simulation allowed us to run many repetitions of the experiments with randomly generated behaviors of the environment, and to replicate exactly the same conditions for the two algorithm.

The simulation was implemented using OMNeT++, an extensible discrete event simulation environment [Varga et al. 2008]. It simulates the arrival of requests from clients, randomly generating requests. The requests arrive at the load balancer of Znn, and are forwarded to one of the idle servers. If no server is idle, then the requests are queued in FIFO order until one server becomes available. Each server processes one

[8]Even though some of these configuration are not valid, namely those where there are 4 servers and the add server tactic is executing, we chose to keep the model simpler, even if it produces some elements in the reachability relationships that will never be used.

request at a time, with a service time distributed with an exponential distribution whose rate of depends on the fidelity of the content being served. In the case of high fidelity, the rate is 1, and for low fidelity the rate is 0.9.

The inter-arrival times between client requests are generated randomly with a rate that changes periodically, matching the possibilities of the environment in the SMG. Every τ units of time, a new arrival rate is selected randomly from the interval $[0, 2]$ with a uniform distribution. That rate is then used to generate exponentially distributed inter-arrivals until the next rate is selected. To be able to simulate the execution of the system with the same random pattern of client requests using each of the two algorithms, the request inter-arrival times and the service times are drawn from two separate random number generators. Thus, we can compare the utility each algorithm achieves when the system faces the same pattern of client requests.

The self-adaptive layer of the simulated system works as follows. The system is monitored by keeping track of request inter-arrival times when a client request arrives, and of the request response times every time a request processing completes. Once every evaluation interval τ, these observations are used to compute their average and standard deviation for the period since the last evaluation. Using the average response time, the fidelity level, and the number of servers in the system, the utility accrued since the last evaluation is computed using the utility functions and preferences shown in Table I.

Next, the adaptation algorithm is used to determine if the system should self-adapt and how. We implemented both the latency-aware algorithm (Algorithm 1) and a non-latency-aware algorithm. The latter is basically the same as the former, except that it does not account for latency other than by considering the adaptation penalty induced by the cost of having a server powered until becomes active.

When the algorithm is run in each evaluation period, it needs to know what is the current configuration of the system, including whether the tactic to add a server is running, and how much progress it has made. This is achieved by maintaining a model of the system configuration that keeps track of the number of servers in the system, and how many of them are active. In addition, the model keeps a list of expected changes in the future. For example, when a new server is added to the system, an expected change reflecting that the server becomes active is recorded with an expected time of λ into the future. In that way, it is possible to determine how much time is needed until the tactic completes. When a server actually becomes active in the simulation, the model of the current system configuration is updated to reflect that change and the corresponding entry is removed from the list of expected system changes.

The predictive model of the environment, $env(x)$ was implemented as an oracle that can predict perfectly the average and variance of the request inter-arrival times for the same horizon used by the algorithm. Although the request arrivals are randomly generated in the simulation, a perfect prediction can still be achieved by generating the inter-arrival times before they are consumed by the simulation.

Implementing the $U(c, e)$ function requires first estimating the average response time for requests when the system has configuration c, and the environment is e. In this case, the relevant properties of the environment are the average and variance of the inter-arrival times. To estimate the average response time needed for the utility calculation, we used queueing theory with a $G/M/c$ queueing model (i.e., for arrivals with a general distribution,[9] exponentially distributed service times, and s

[9]We chose to use a model for a general distribution of arrivals since: (i) although arrivals are generated with an exponential distribution, the rate parameter of the distribution is changed periodically, and (ii) the queueing model is for steady-state behavior and does not account for any backlog of requests that could have

servers) [Gross et al. 2011]. Once the average response time is estimated in this way, the utility is estimated using the utility functions and preferences shown in Table I.

After the adaptation algorithm has determined how the system has to be changed, the execution of the adaptation tactics is carried out by adding or removing servers, and changing the fidelity as needed. The standard queuing components of OMNET++ were modified to support this dynamic reconfiguration. Furthermore, the server component was modified to simulate the latency of enlisting a server.

5.3. Results

We ran the simulation with the same parameters used for the SMG analysis, as described in 4.3. The horizon used for the algorithms was computed so that if the system was running with one server, it had a horizon large enough to be able to compute the effect of adding the three remaining servers. For that reason, the horizon was calculated as $3\frac{\lambda}{\tau}+1$, the number of periods needed to enlist three servers plus one more period to consider the impact on utility of the change. For each combination of parameters, the simulation was run 1000 times to obtain the statistics shown in Table III. On average, the latency-aware algorithm outperformed the non-latency-aware one. The LA algorithm obtained on average about 5% more utility when the tactic latency was equal to the evaluation period, and 10% for latencies two and three times larger than the evaluation period. The standardized effect size measure statistic $\hat{A}_{12}$ [Arcuri and Briand 2012] shows that LA outperforms NLA 66% to 81% of the times, depending on the parameters. For several combinations of parameters, the minimum percentual utility difference $\Delta U(\%)$ was negative, meaning that NLA did better. This is due to a limitation of the queueing model used by the algorithms to estimate the response time of different configurations, because it computes the steady-state response time, and, therefore, ignores the effect of arrival spikes that may leave a backlog of arrivals to be processed in later periods. The LA algorithm avoids adaptation when there are transient increases in load if the cost of enlisting a server will be higher than the negative impact of not adding it. Because of the limitation of the queueing model, it sometimes underestimates that negative effect. Since the NLA algorithm does not account for the latency of the tactic, it is more prone to add servers, and that gives it an advantage in these cases. These situations were not very common in our experiment runs, as indicated by the 10% quantile, which, except for the cases with the lowest tactic latency, was positive. Furthermore, it is worth noting that this is a limitation of the $U(c,e)$ function used by the algorithm, and not a problem with the algorithm itself.

Table III: Simulation results for Znn

MAX_TIME (s)	Latency (s)	Latency-Aware			Non-Latency-Aware			$\hat{A}_{12}$	$\Delta U(\%)$			
		min.	avg.	max.	min.	avg.	max.		min.	10% quant.	avg.	max.
100	TAU	39.18	67.29	84.41	33.80	62.63	84.49	0.66	-27.15	-0.65	6.73	31.32
	2*TAU	44.66	69.33	84.55	36.33	62.31	83.20	0.73	-23.86	3.10	10.34	37.69
	3* TAU	48.05	69.40	84.55	31.14	62.48	83.20	0.72	-0.88	3.12	10.24	38.66
200	TAU	81.99	133.20	167.20	82.48	125.00	156.90	0.69	-15.63	-0.96	5.98	21.70
	2*TAU	105.90	138.10	167.20	80.46	124.40	160.00	0.81	-7.82	4.89	10.05	30.53
	3* TAU	106.20	138.40	167.20	85.81	124.70	160.00	0.81	0.00	4.85	10.01	28.32

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have described an analysis technique based on model checking of stochastic multiplayer games that enables the quantification of the potential benefits

remained in the system from a previous period with higher traffic intensity. Hence, we found the general distribution model was a better fit.

that different types of algorithms for decision making in adaptation can yield. We have shown how this technique can be used in the context of comparing proactive adaptation algorithms that consider information about tactic latency for decision-making, against those that do not account for it. We have used Znn.com to illustrate our approach.

Our results show that latency-aware proactive adaptation always performs better than non-latency-aware adaptation both in the worst- and best-case scenarios, with progressively increasing improvements with higher tactic latencies. Moreover, results indicate that not considering latency information in decision-making can potentially inhibit the selection of tactics available in the adaptation model which could help improve the performance of adaptation algorithms.

A current limitation of the approach is that its scalability is limited by PRISM-games, which currently uses explicit-state data structures and is to the best of our knowledge the only tool supporting model-checking of SMGs. In our case, the largest SMG model employed for Znn has of the order of 10^6 states, whereas the results presented in [Chen et al. 2013b] show that the current version of the tool can handle models of up to 10^7 states in a common desktop PC. However, the authors of PRISM-games are developing a symbolic (BDD-based) version of the tool that will improve scalability.

We have also proposed a latency-aware proactive adaptation algorithm that is able to exploit predictions about the future behavior of the environment. We have compared our algorithm against the proactive algorithm presented in [Poladian et al. 2007], which does not consider latency, showing that latency-aware adaptation achieves higher utility.

Regarding future work, we plan to instantiate our adaptation analysis technique in different contexts. In particular, we are working on applying this approach to self-protecting systems, studying how different adaptation alternatives can minimize the damage that an attacker can inflict. We also aim at refining the approach to do runtime synthesis of proactive adaptation strategies based on SMGs. Concerning latency-aware adaptation, we aim at exploring how tactic latency information can be further exploited to attain better results both in proactive and reactive adaptation (e.g., by parallelizing tactic executions). We will also generalize the algorithm to consider multiple tactics with different latency, as well as prediction and tactic latency uncertainty. Moreover, we will implement our algorithms in Rainbow/Znn.

REFERENCES

Rajeev Alur and others. 2002. Alternating-time temporal logic. *J. ACM* 49, 5 (2002).

Andrea Arcuri and Lionel Briand. 2012. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* (2012). DOI:http://dx.doi.org/10.1002/stvr.1486

Andrea Bianco and Luca de Alfaro. 1995. Model Checking of Probabalistic and Nondeterministic Systems. In *FSTTCS (LNCS)*, Vol. 1026. Springer.

Víctor Braberman and others. 2013. Controller Synthesis: From Modelling to Enactment. In *ICSE*. IEEE.

Radu Calinescu and others. 2011. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Software Eng.* 37, 3 (2011).

T. Chen and others. 2013a. Automatic Verification of Competitive Stochastic Systems. *Form Method Syst Des* 43, 1 (2013).

T. Chen and others. 2013b. PRISM-games: A Model Checker for Stochastic Multi-Player Games. In *Proc. of TACAS'13 (LNCS)*, Vol. 7795. Springer.

Taolue Chen and Jian Lu. 2007. Probabilistic Alternating-time Temporal Logic and Model Checking Algorithm. In *FSKD*, Vol. 2.

S.W. Cheng and others. 2009a. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *SEAMS*. IEEE.

Shang-Wen Cheng and others. 2009b. Improving Architecture-Based Self-Adaptation through Resource Prediction. In *SEfSAS*. LNCS, Vol. 5525. Springer.

Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012).

R.M. Chiulli. 1999. *Quantitative Analysis: An Introduction*. Taylor & Francis.

V. Forejt and others. 2011. Automated Verification Techniques for Probabilistic Systems. In *SFM (LNCS)*, Vol. 6659. Springer.

Anshul Gandhi and others. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4 (2012).

David Garlan and others. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (2004).

RobertP. Goldman and others. 2003. Managing Online Self-adaptation in Real-Time Environments. In *Self-Adaptive Software: Applications*. LNCS, Vol. 2614. Springer.

D. Gross and others. 2011. *Fundamentals of Queueing Theory*. Wiley. http://books.google.com/books?id=K3lQGeCtAJgC

Julia Hielscher and others. 2008. A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing. LNCS, Vol. 5377. Springer.

Wiebe Van Der Hoek and Michael Wooldridge. 2003. Model Checking Cooperation, Knowledge, and Time - A Case Study. In *Research in Economics*.

Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. The MIT Press.

J. Kramer and J. Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *FOSE*.

Steve Kremer and Jean-Francois Raskin. 2001. A Game-Based Verification of Non-repudiation and Fair Exchange Protocols. In *CONCUR 2001*. LNCS, Vol. 2154. Springer.

M. Kwiatkowska and others. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV (LNCS)*, Vol. 6806. Springer.

Andreas Metzger and others. 2013. Accurate Proactive Adaptation of Service-Oriented Systems. In *ASAS*. LNCS, Vol. 7740. Springer.

David Musliner. 2001. Imposing Real-Time Constraints on Self-Adaptive Controller Synthesis. In *Self-Adaptive Software*. LNCS, Vol. 1936. Springer.

Peyman Oreizy and others. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intell. Syst.* 14 (1999), 9. Issue 3.

Vahe Poladian and others. 2007. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In *SASO*.

András Varga and others. 2008. An overview of the OMNeT++ simulation environment. In *Simutools*. ICST.

Chen Wang and J-L Pazat. 2012. A Two-Phase Online Prediction Approach for Accurate and Timely Adaptation Decision. In *SCC*.

Xu Zhang and Chung-Horng Lung. 2010. Improving Software Performance and Reliability with an Architecture-Based Self-Adaptive Framework. In *COMPSAC*.

Received February 2007; revised March 2009; accepted June 2009